Information Flow Control and Taint Analysis with Dependence Graphs

Jens Krinke
FernUniversität in Hagen
Hagen, Germany

1. Introduction

Ensuring that the integrity of critical computation is not violated by untrusted code or the confidential data is protected is a complex problem for current software systems. We can observe two main directions to approach the problems:

- For critical system, formal approaches are needed. One is (static) *information flow control* which analyzes the software to check if it conforms to some security policy. An example is *noninterference*: secret information does not influence the publicly observable behavior of a system.
- Many informal approaches can be subsumed under *bug detection*. A violation of some security policy can be regarded as a bug and therefore many bug detection approaches do some kind of taint analysis. Data from untrusted sources (e.g. the user) is tainted and is not allowed to reach exploitable functions like system calls vulnerable to buffer overruns.

There is a large overlap in both directions. Let's consider taint analysis as described in the later direction: Taint analysis is just a special case of information flow control and noninterference (tainted information does not influence vulnerable parts of the system). Moreover, it is a very simple form as it just has a security policy with two levels "tainted" and "untainted".

There is a major difference in both directions: The first is conservative while the later is optimistic. This means that the formal approaches do not allow false negatives: If the noninterference check does not reveal a violation, it is guaranteed that the analyzed code does not contain leaks. On the other hand, bug detecting approaches prefer an optimistic approach that allows false negatives. This has two reasons: First, this reduces the amount of false positives, and second, practicable solutions (as in tools) often require unsound approaches.

It is not surprising that both directions have almost disjunct research communities. For example, the recent comprehensive survey by Sabelfeld and Myers [11] about information flow security does include 147 references, but none of them is related to taint analysis for bug detection. However, a closer examination reveals that both communities can benefit from each other. Especially as both communities make heavy use of program analysis for their approaches.

For information flow control, many approaches use program analysis in terms of special type systems that ensure noninterference or other security policies for well-typed programs. Such type systems are usually extensions or modifications of real world languages, examples are Jif by Myers et al. [6, 7], similar to Java, and Flow Caml by Simonet and Pottier [10, 12], similar to Objective Caml. However, as Zdancewic [15] has observed, information-flow based enforcement mechanisms have not been widely used.

In contrast, taint analysis for bug detection is more and more used for the analysis of real systems. Many of such approaches are based on a long record of research on program analysis for optimization and reengineering [1]. Usually, type systems are only one of the used analyses (most often used for efficient pointer analysis) and other, more complex representations are used. For example, the taint analysis approaches from Pistoia et al. [9] use program slicing [13] or the one from Livshits and Lam [4, 5] that uses IPSSA, an interprocedural variant of gated SSA [2, 8]. Wilander discusses some more tools.

It seems that the bug detection approaches are somewhat more successful than the information flow control approaches. From our point of view, this is related to major difference that has not been pointed out by Zdancewic as a challenge [15]: Information flow control approaches and tools like Jif and Flow Caml rely on dedicated languages that support information flow control while the bug detection approaches analyze real world programs in standard languages like C or Java with additional security-related information provided. This enables security analyses on already existing software, whereas in information flow control approaches one needs to reimplement the system.

In the following we will present our approach for information flow control that is based on dependence graphs, using results from both major directions discussed above. It will enable more flexible security policies than taint analysis without for real world languages like C or Java, thus having the advantages of both directions.

Published in the Proceedings Third International Workshop on Code Based Software Security Assessments (CoBaSSA), 2007 in Vancouver, Canada.

2. Security levels and declassification

The simple two-level security policy of taint analysis is sufficient for simple problems, but in practice one wants more detailed information about security levels of individual statements. Thus theoretical models for IFC utilize a *lattice* $\mathcal{L} = (L, \sqcup, \sqcap)$ of security levels, the simplest consisting just of two security levels *High* and *Low*. We provide a specification option for the lattice, and an option to mark some (or all) statements with their security level. The approach is based on the program dependence graph representation [3], Program statements or expressions are the graph nodes. A data dependence edge $x \to y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependence edge $x \to y$ means that the mere execution of y depends on the value of the expression x (which is typically a condition in an if- or while-statement). A path $x \to^* y$ means that information can flow from x to y; if there is no path, it is guaranteed that there is no information flow. In particular, all statements influencing y (the so-called *backward slice*) are easily computed as $BS(y) = \{x \mid x \to^* y\}$. If there is no PDG path from a to b, it is guaranteed there is no information flow from a to b. This is true for all information flow which is not caused by hidden physical side channels such as timing leaks.

The security level of statement with PDG node x is written $S(x)$, and confidentiality requires that an information receiver must have at least the security level of any sender. In PDGs, this implies $\forall y \in pred(x) : S(x) \geq S(y)$ which ensures $S(y) \rightsquigarrow S(x)$. The dual condition for integrity is $\forall y \in pred(x) : S(x) \leq S(y)$ However, this assumes that every statement resp. node has a security level specified, which is not realistic. We want to specify *provided* as well as *required* security levels not for all statements, but for certain selected statements only. The provided security level specifies that a statement sends information with the provided security level and the required security level specifies that only information with a *smaller* security level may reach that statement. The provided security levels are defined by a partial function $P : N \to L$, where N is the set of nodes resp. statements of the programs. Thus, $l = P(s)$ specifies the statement's security level. The required security levels are defined similarly as a partial function $R : N \to L$. Thus, $P(s)$ specifies the security level of the information generated at s and $R(s)$ specifies the maximal allowed security level of the information reaching s. Information with security level l that is generated at some node x in the dependence graph, is propagated along the dependences and should not reach another node a which has a required security level which is smaller than l. Thus a program represented as a dependence graph does not violate confidentiality, iff

$$\forall a \in \mathrm{dom}(R) : \forall x \in BS(a) \cap \mathrm{dom}(P) : P(x) \leq R(a)$$

```
1  class PasswordFile {
2    private String[] names/*P:confidential*/
3    private String[] passwords; /*P:secret*/
4    public boolean check(String user,
5        String password /*P:confidential*/) {
6      boolean match = false;
7      for (int i=0; i<names.length; i++) {
8        if (names[i]==user
9            && passwords[i]==password) {
10         match = true;
11         break;
12       }
13     }
14     return match;  /*R:public*/
15   }
16 }
```

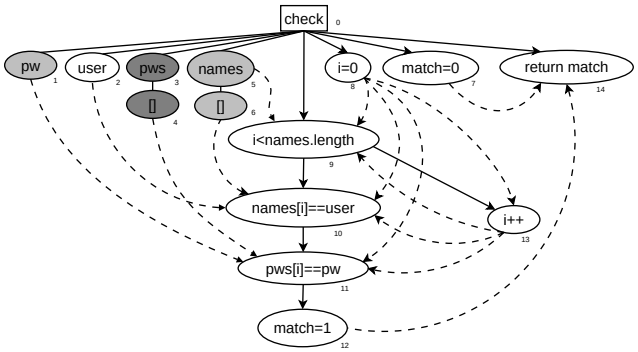

Figure 1. A Java password checker with its PDG

i.e. the backward slice from a node a with a required security level $R(a)$ must not contain a node x that has a higher security level $P(x)$.

Usually, the number of nodes that have a specified security level is low, e.g. points of output. Therefore, the above criterion can easily be transformed into an algorithm that checks a program for confidentiality:

PDG-Based Confidentiality Check. *For every node in the dependence graph that has a required security level specified, compute the backward slice, and check that no node in the slice has a higher provided security level specified.*

Checking each node separately allows a simple yet powerful diagnosis in the case of a security violation: If a node x in the backward slice $BS(a)$ has a provided security level that is too large ($P(x) > R(a)$), the responsible nodes can be computed by a chop $CH(x, a)$. The chop computes all nodes that are part of path from node x to node a, thus it contains all nodes that may be involved in the propagation from x's security level to a.

As an example, consider the PDG for a password checking program (Figure 1) . We choose a three-level secu-

rity lattice: *public*, *confidential*, and *secret* where *public* $\rightsquigarrow$ *confidential* $\rightsquigarrow$ *secret*. The list of passwords is *secret*, thus $P(3) = P(4) =$ *secret*. The list of names and the parameter `password` is *confidential*, because they should never be visible to a user. Thus, $P(1) = P(5) = P(6) =$ *confidential*. Figure 1 shows the annotated PDG: The security levels are depicted through white for *public*, light gray for *confidential*, and gray for *secret*. According to the criterion, we require that no confidential or secret information flows out of the method, thus we require the return statement to have a required security level of *public* ($R(14) =$ *public*). A backward slice for node 14 will reveal that nodes 1 and 3–6 are included in the slice and have a higher security level, thus a security violation is revealed.

In practice the above approach is too simple because in some situations one might accept that information with a higher security level flows to a "lower" channel. A typical example is the password checking method presented earlier: The result of the method will eventually be used to access the user's private data or to output an error message that the login was not successful. Of course, that areas will not have the same security level (*secret*) as the list of passwords. *Declassification* allows to lower the security level of incoming information at specified points. In the example, declassification would reduce the security level at the return node 14 to security level *public* such that the result of the password check can be used in low security areas.

We model declassification by specifying certain PDG nodes to be declassification nodes: Let D be the set of declassification nodes. A declassification node $x \in D$ has to have a required and a provided security level: $r = R(x)$ and $p = P(x)$. Information reaching x with a maximal security level r is lowered (declassified) down to p. Now a path from node y to a with $P(y) > R(a)$ is not a violation, if there is a declassification node x on the path with $P(y) \leq R(x)$ and $P(x) \leq R(a)$ (assuming that there is no other declassification node on that path).

The above slicing solution no longer works with declassification, as information flow with declassification is no longer transitive and slicing is based on transitive information flow. This can easily be solved: First, we compute the backward slice for every node a with a required security level specified. By definition of a slice, no node outside this slice can have any influence on a. The subsequent analysis (see below) will only consider nodes and edges that are part of this slice, i.e. the dependence graph is reduced to the subgraph represented by the initial slice for a. This avoids spurious dependencies and false alarms caused by potential security violations outside the backward slice of a.

The analysis will then compute the actual required security level for every node in the (sliced) program dependence graph by a backward analysis. The actual required security level of a node is the maximal security level that may reach the node without causing a security violation at the criterion node under observation.

The actual incoming (required) security level $S_{\text{IN}}(x)$ for a statement x is computed from the outgoing security levels of its successors $y \in succ(x)$:

$$S_{\text{IN}}(x) = \begin{cases} \top & \text{if } succ(x) = \emptyset \\ \bigsqcap_{y \in succ(x)} S_{\text{OUT}}(y) & \text{otherwise} \end{cases}$$

At nodes without declassification, the outgoing security level is simply the incoming security level: $S_{\text{OUT}}(x) = S_{\text{IN}}(x)$. At declassification nodes $x \in D$ with a declassification from $R(x)$ down to $P(x)$, S is replaced with the new required level: $S_{\text{OUT}}(x) = R(x)$. Thus

$$S_{\text{OUT}}(x) = \begin{cases} R(x) & \text{if } x \in D \\ S_{\text{IN}}(x) & \text{otherwise} \end{cases}$$

These equations are data flow analysis equations and can be iteratively solved using a standard algorithm, with a proper initialization of S_{OUT}. The initialization is done based on the criterion node a under observation, i.e. it is checked that no security violation occurs due to $R(a)$:

$$S_{\text{OUT}}(x) = \begin{cases} R(a) & \text{if } x = a \\ R(x) & \text{if } x \in D \\ \top & \text{otherwise} \end{cases}$$

Due to the monotonicity of the computation and the limited height of the security level lattice, a minimal fixed point for S_{IN} is guaranteed to exist and can be computed using a standard iteration. The computed S_{IN} have then to be checked for confidentiality:

Confidentiality Check With Declassification. *For every node a in the dependence graph that has a required security level specified which is not a declassification node, compute the incoming security levels $S_{\text{IN}}(x)$ of all statements x in its backward slice and check the following property:*

$$\forall x \in dom(P) \cap BS(a) : P(x) \leq S_{\text{IN}}(x) \qquad (1)$$

Thus, for any $l = P(x)$ such that $l \not\leq S_{\text{IN}}(x)$ we have a confidentiality violation at x because $l \not\rightsquigarrow S_{\text{IN}}(x)$ (the security level l is not allowed to influence the required level of $S_{\text{IN}}(x)$. Note that it is $\not\leq$ and not $>$ because l and $S_{\text{IN}}(x)$ might not be comparable. Because the slice $BS(a)$ has been computed first, the confidentiality check can be done during the dataflow analysis: If the current node has a provided security level $P(x)$, a computed required level $S_{\text{IN}} \not\geq P(x)$ is a violation. Declassification nodes themselves are not considered as information sinks in the above check, even though they have to have a required security level.

Let us return to the example in Figure 1 and assume $R(14) =$ *public*. The computation to check confidentiality for criterion node 14 will start with a backward slice $BS(14)$. Because node 14 can be reached from every node

in the PDG, the slice will contain the complete PDG, thus it is not reduced. The subsequent computation of the actual required security levels will result in $S_{\text{IN}}(x) = \textit{public}$ for all nodes x of the example. The confidentiality check will reveal violations at nodes 1 and 3–6 because for these nodes, the specified provided level is higher than the computed actual required.

Now assume the node 14 is a declassification node $\textit{secret} \rightarrow \textit{public}$: $14 \in D$, $R(14) = \textit{secret}$, $P(14) = \textit{public}$. The computation of the actual required security levels will result in $S_{\text{IN}}(x) = \textit{secret}$ for $x < 14$. The confidentiality check will no longer reveal a security violation, which may be desirable depending on the security policy, since only a negligible amount of information leaks from password checking.

3. Taint analysis

The above presented approach has been implemented and we are currently evaluating it for confidentiality security polices. While preparing the evaluation for integrity security policies (to which taint analysis belongs), we discovered a disadvantage of the proposed approach. For taint analysis, we are able to specify the sources of untrusted data (e.g. user inputs) as with a provided security level of *tainted* and to specify the vulnerable functions to have a required security level of *untainted*. However, many operations or functions do a declassification from *tainted* down to *untainted*. For example, consider taint variable analysis for buffer overrun protection. The call of `snprintf` from the C-library prints its arguments to a buffer. However, it is guaranteed by `snprintf` that the buffer will not overrun. Thus, `snprintf` declassifies a security level of *tainted* in its arguments down to *untainted* in the buffer. Such a declassification can easily be specified within our approach by specifying the assignment to the buffer inside `snprintf` as declassification node.

However, there are other operations that turn tainted variables into untainted ones. An example is a simple check of the buffer with the user input against a maximal size ("`if (length(s) < MAX)`"). In the true branch, `s` is *untainted* and in the false branch, it is *tainted*. Such a declassification cannot easily be specified within our approach: Any use of variable `s` inside the true branch will have a direct data dependence to the *tainted* assignment to `s` and will result in a generated level of *tainted*.

We are currently investigating more flexible way to specify declassifications. One approach would be to use Wilander's pattern matching on dependence graphs [14]. For the above example, the declassification would be specified as "a check of the length of a variable against a maximal size will declassify all accesses to the variable in the true branch to *untainted*". As SSA form is needed here to prevent that the check and the access are different instances of the same variable, another approach that is investigated by is to use gated SSA form.

References

[1] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, November/December 2004.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, 1991.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[4] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, 2003.

[5] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.

[6] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[7] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. http://www.cornell.edu/jif, 1999.

[8] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 257–271, 1990.

[9] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP 2005 – 19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, 2005.

[10] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Prog. Lang. Syst.*, 25(1):117–158, 2003.

[11] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[12] V. Simonet. Flow caml. http://cristal.inria.fr/ simonet/soft-/flowcaml/.

[13] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[14] J. Wilander and P. Fak. Pattern matching security properties of code using dependence graphs. In *Proceedings of the First International Workshop on Code Based Software Security Assessments*, pages 5–8, 2005.

[15] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.